Copper Design Document

Table of Contents

[General Overview 3](#_Toc350536779)

[Copper Memory 3](#_Toc350536780)

[Registers 3](#_Toc350536781)

[CPU Accessable 3](#_Toc350536782)

[Internal (Inside the Copper, not accessable to Z80) 4](#_Toc350536783)

[Using the COPPER Coprocessor 4](#_Toc350536784)

[AutoRestart (The simplest configuration) 5](#_Toc350536785)

[Manual Restart (Mainline) 5](#_Toc350536786)

[ISR Driven 6](#_Toc350536787)

[Instructions 6](#_Toc350536788)

[0 Instructions 6](#_Toc350536789)

[1 Instructions 11](#_Toc350536790)

[Internal Functional Overview 11](#_Toc350536791)

[Copper Compiler 12](#_Toc350536792)

[Script Language 12](#_Toc350536793)

[SETADDR 12](#_Toc350536794)

[FILLV 12](#_Toc350536795)

[WAITNSS, WAITNSL 13](#_Toc350536796)

[WAITS 13](#_Toc350536797)

[STOP 13](#_Toc350536798)

[DMAS, DMAL 13](#_Toc350536799)

[SKIP 13](#_Toc350536800)

[Example.cop 13](#_Toc350536801)

# General Overview

The Copper Coprocessor is able to access all VRAM and REGS within the VideoChip, at full VGA clock speeds. It can manipulate data on a per-scanline basis. It is similar to the Amiga’s Copper device.

# Copper Memory

The Copper requires BRAM(s) for its’ instruction list(s).

Each instruction is between 1 to 3 bytes long. With one BRAM @ 2K bytes, that gives us about 900 of the smaller instructions per BRAM. A video bank is up-to 8K (4 BRAMS), so that could comfortably hold 4000 copper instructions. To build double-buffered Copper list, half the above numbers. With 240 scanlines, and a double-buffered copper list in an 8K BRAM, that’s about 34 bytes per scanline, or about 16 instructions per scanline.

## Registers

### CPU Accessable

#### COPPER\_START\_ADDR\_H/L

This register allows the CPU to control what location the Copper starts executing at next time it is restarted. It also allows the CPU to build a new list at a different address region while displaying the current list.

* xxxAAAAA AAAAAAAA  
  A: Address within the COPPER\_START\_BANK, where the first instruction is located.

#### COPPER\_START\_BANK

This register allows the CPU to specify which bank the Copper will look in, when fetching instructions after the next restart.

* xxxxBBBB  
  B: VRAM Bank# where Copper instructions are stored.

#### COPPER\_CONTROL

* EAxxxxR  
  E: Enable Copper (1=Copper can execute, 0=Copper is completely disabled).   
  NOTE: If you enable Copper without issuing a RESTART, then the Copper will immediately begin to execute from its’ current (internal) instruction pointer location.   
  NOTE: If you had previously stopped the copper, by setting Enable to 0, then the IP *could* be pointing to the middle of an instruction, and thus would begin to execute garbage when Enable is set back to 1. Therefore, the only time it makes sense to set Enable to 1, without also writing a 1 to Reset as well, is if the copper had previously become disabled by executing a STOP instruction.

A: AutoRestart (1=Copper will automatically restart on every VBLANK. 0=Copper will restart only when the CPU assigns a 1 to the R bit.  
R: Restart (1=RESTART, 0=no change)  
(Write: 1=go to COPPER\_START\_ADDR, 0=no change. Read: 0)  
Writing a 1 to this address will cause the Copper to immediately restart execution from the address specified at COPPER\_START\_ADDR. Reading from this flag always results in the value 0.

For the Z80 to programmatically instruct the Copper to stop, set the Enable flag to false. There is also a Copper instruction called STOP. If the Copper executes it, then the ENABLE flag will return to 0 and the Copper will not execute any more instructions. If the CPU and Copper ever try to manipulate the same copper flag at the same time, the CPU always wins (because the copper register logic is designed this way). If the CPU and the copper try to manipulate the same VRAM byte at the same time, the copper wins (because the copper memory access is delayed by one cycle while the CPU is busy).

### Internal (Inside the Copper, not accessable to Z80)

#### COPPER\_IP\_ADDR

The VRAM address which the Copper will fetch its next instruction from.

#### COPPER\_IP\_BANK

The VRAM bank which the Copper will fetch its next instruction from.

# Using the COPPER Coprocessor

The Copper can be configured for use in a wide variety of ways.

A typical Copper implementation is a list of instructions that either manipulate VRAM or wait for particular scanline(s).

E.g. To create a screen that appears to be split into two regions, black and blue:

* Set the background color to black.
* Wait for scanline 120 (half way down the screen)
* Set the background color to blue.
* Wait for scaline 255 (end of instructions, because scanline 239 is actually the last one the Copper will see, before the counter starts over).

The make the copper execute, there are a few options:

1. Use an ISR on the CPU that sets ENABLE and RESET to true on every VBLANK. This also gives the CPU a chance to customize the copper list before starting it on each frame.
2. Set the ENABLE, RESET, and AUTO\_RESTART flags of the copper. This sets the copper into a completely automatic mode, where it restarts itself on every vblank.

A more advanced system could have multiple Copper lists. Each list could perform a DMA-related task, such as scrolling the map data up, down, left, or right by one cell, animating some cell data via DMA’s, etc. They could also set and clear a byte chosen somewhere in VRAM that represents that they are busy or idle. The CPU’s ISR could then trigger each required list, each time waiting for the busy flag to clear, before triggering the next list. Until finally, triggering the main on-screen display-list as normal. The ISR could then go on to perform other non- Copper tasks such as updating music, input, gameplay, etc. while the copper is performing its various DMAs.

The Copper does not steal any cycles from the CPU. It runs completely in parallel. The CPU can steal a very small number of cycles from the Copper, if it reads or writes VRAM. However, the copper uses the VRAM bus on only half of the available cycles, and it can resync with the CPU if a collision occurs, so it’s unlikely to be affected noticeably, even by very heavy CPU VRAM access.

Here are a few examples of typical configurations:

## AutoRestart (The simplest configuration)

The simplest Copper configuration involves a Copper list that will automatically execute from the top, starting on every VBLANK, completely automatically.

* Setup an instruction list for the Copper to execute.   
  For example, to create a split in the background color half way down the screen…
  + Set the background color to BLUE.
  + Wait for Scanline # 120 (half way down the screen)
  + Set the background color to BLACK.
  + Wait for Scanline # 255   
    *(which never happens. Thus, the Copper is at the end of its’ instruction list).*
* Assign the start-address of the Copper-list, into COPPER\_START\_BANK, COPPER\_START\_ADDRH, COPPER\_START\_ADDRL.   
  *Eg. If you chose the start of BG2 CellData, the bank would be the BG2 CellData Bank, and the start address would be 0 (the first byte in that bank of VRAM).*
* Set the ENABLE, AUTO\_RESTART, and RESET flags of the copper, all in a single write operation. This instructs the copper to begin execution from the IP you had previously written, and to restart itself on every VBLANK.
* The Copper will automatically execute the above list of instructions, starting from the top, beginning on every VBLANK, causing the top half of the screen to appear blue, and the bottom half to appear black.
* The CPU could then perform some tricks by writing a new scanline number into the copper’s wait instruction, to shift where the color-change occurs. Or it could change the color.

## Manual Restart (Mainline)

The Copper can assist in moving VRAM memory around, far faster than the CPU could move the same data. And, the Copper can do it while the CPU isn’t looking (allowing the CPU to parallelize by executing other tasks). Typical uses might be…

* Scrolling of the map data horizontally or vertically by a full byte. The CPU would then jump in afterward and just fix-up the newly exposed edge.
* Copying frames of sprite animation from some off-screen area of VRAM, to an on-screen sprite image.
* Filling a BG with a specific color.

Operations of these sort, are not typically desired on every single frame. The Copper can be manually triggered at any time by the CPU. An above operation could be performed, on-demand, by turning off the Copper’s “AutoRestart” flag, then setting up the COPPER\_START address and bank, and manually triggering an ENABLE and RESTART only when one of these operations is actually needed.

To detect when the Copper has completed, choose an unused byte in VRAM, and use it as a “DONE” flag. The first 2 Copper instructions should clear the flag (load the DST register, then write a 0 into it), and the final 3 instructions should set the flag before finally waiting for scaline 255, as is usual to end a Copper list.

(load the DST register, write a 1 into it, and then wait for scanline 255)

## ISR Driven

The most powerful configuration allows you to use the CPU to control the Copper’s execution from an ISR. The ISR offers you synchronization with the display. It also offers you time between the start of VSYNC and the top of the display, for you to execute whatever Copper lists you may wish, such as course scrolling of the BG map, or DMA’ing new sprite images into onscreen VRAM, before finally starting the on-screen Copper list.

## Instructions

### 0 Instructions

#### 000 Instructions

##### FILLV: Fill area beginning at DST with VALUE, (1…32 bytes)

* 000NNNNN VVVVVVVV (OPCODE, PARAM)
* N: 0..31 (1..32 bytes to fill starting at DST)
* V: The value to write

NOTE: If you, at runtime, want to adjust the value to 0, then replace the instruction with a SKIP 1 instruction.

#### 001 Instructions

##### Unused 001 Instructions

Unused Instructions: 00100000.. 00101110

##### SETADDR: Set SRC/DST Address

* 00101111 TMMMxxBB BBBAAAAA AAAAAAAA (OPCODE, PARAM1, PARAM2, PARAM3)
* B: VRAM Bank#
* T: Target 0=SRC, 1=DST
* M (Mode):  
   000=No change.  
   001=post-INC.  
   010=post-DEC.  
   011=post-INC AND SWAP NIBBLES (works on SRC only).  
   100=(not used)  
   101=post-INC AND RESTORE to initial value after instruction.  
   110=post-DEC AND RESTORE to initial value after instruction.  
   111=post-DEC AND SWAP NIBBLES (works on SRC only).
* A: Address (0..8191)

NOTE: The ‘post-INC and restore’ mode allows a DMA or FILL instruction to change the SRC or DST register value throughout the operation, and then return it to the pre-instruction value as soon as the instruction completes. This is handy, for things like setting the same palette entries to new colors on each scanline.

NOTE: The ability to swap nibbles, combined with INC & DEC together, allows for reversing image data horizontally.

NOTE: The Bank and the address are all run into each-other, so that they make what feels like a linear address, for situations where data traverses multiple VRAM banks (eg. sprite data)

##### WAITNSS: Wait for N scanlines (1..8)

Since this is a relative scanline (wait for 1..8 scanlines below the current scanline), this allows an entire Copper-list to be vertically scrolled easily. This also offers the numbers 1 and 8, but in only one byte, so scanline or cell-based effects could be coded in a smaller number of instruction bytes.

* 00110NNN (OPCODE)  
  N: 0..7 (wait for 1..8 scanlines)

##### WAITNSL: Wait for N scanlines (0..255)

Since this is a relative scanline (wait for 0..512 scanlines below the current scanline), this allows an entire Copper-list to be vertically scrolled easily. This instruction also offers 0, meaning the wait will not occur.

* 0011100N NNNNNNNN (OPCODE, PARAM)  
  N: 0..511 (wait for 0..511 scanlines)

##### IF: Conditional Flow & GOTO

[NOT IMPLEMENTED]

This instruction is capable of creating conditional flow within the copper list. It also allows for the IP to travel backward in the list.

**IF [signed|unsigned] \*SRC&<mask>[==|!=|<|<=|>|>=|ALWAYS|?] <value> GOTO label**

NOTE: There are now unused instruction opcodes in the 001 region.

* TODOTODO MMMMMMMM VVVVVVVV SNCCLLLL LLLLLLLL (OPCODE, PARAM1, PARAM2, PARAM3, PARAM4)  
  M: A mask to AND the value found at SRC with.  
  V: A value to compare the masked result against.  
  N: 0=use comparison result, 1= use NOT comparison result  
  S: 0=Unsigned comparison, 1=Signed comparison   
  C: A comparison to use, as follows:
  + 00: ALWAYS
  + 01: ==
  + 10: >
  + 11: >=

L: A signed offset from the end of this instruction, to jump to (-2048..2047)

##### WAITX: Wait for pixel #N (0..511)

*This instruction has limited usefulness in the current video-chip design, since the scanline is rendered using multiple passes. The instruction waits until the BG render pass has visited the specified pixel. It cannot be used, for sprite location or palette tricks, only BG related palette tricks can be performed.*

This instruction can wait for an approximate x coordinate across the scanline.

It is approximate, because the copper is affected a little bit, by CPU activity against VRAM. It also takes a few cycles to execute each copper instruction, and a pixel is 1 cycle long when rendering the BG’s, but a copper instruction’s cycle count can vary. This instruction is good for things like changing the foreground text color, part way across a screen, so long as there is a character or two of white space between the color regions.

* 0011101N NNNNNNNN (OPCODE, PARAM)  
  N: 0..511 (wait for pixel# 0..319 on this scanline)
* NOTE: Waiting for a pixel# above 319 will cause the copper to wait forever, since there are no pixels in that area (320..511).

##### WAITS: Wait for scanline #N (-256..255)

This instruction can wait for a very specific scanline number. It can wait for scanlines within the VBLANK (the negative values), as well as on-screen scanlines (0..239).

* 0011110N NNNNNNNN (OPCODE, PARAM)
* NOTE: Waiting for scanline#255 is similar to STOP, except that the copper’s enable flag is not set to 0. Thus, the copper will still be auto-restarted by a VBLANK. *The copper compiler has a virtual instruction, called WAITVBL, which actually performs a WAITS 255.*

##### Extended Instructions

This value is reserved for future instruction-set expansion (it makes multibyte opcodes a possibility, as well).

* 00111110 XXXXXXXX […] (EXTENDED\_INSTRUCTION, OPCODE, PARAM1, PARAM2, …)  
  X: The extended instruction, followed by whatever parameters are required.

#### PIXELSHIFT (Extended Instruction)

[NOT IMPLEMENTED]

This instruction is capable of shifting a block of pixels 1 to the left, or right, including optional modes which understand the layout in memory, of sequential chars or sprites.

**PIXELSHIFT DST <Left|Right> <LINEAR|CHAR|SPRITE> 1..128** bytes= 2..256 pixels

* 00111110 000000LL DNNNNNNN (EXTENDED\_INSTRUCTION, OPCODE, PARAM)  
  L: Pixel Layout
  + 00=Linear (no bytes per row organization, simply access sequential bytes)
  + 01=Char (4 bytes per row, skipping 8\*4=32 bytes to find start of next row)
  + 10=Sprite (8 bytes per row, skipping 8\*8=64 bytes to find start of next row)
  + 11=(not used)

D: 0=Left, 1=Right  
N: 0..127 = 1..128 bytes (2..256 pixels)

##### Internal Algorithm

Internally, the algorithm uses a linear count to traverse the area to be shifted. The count then goes through some massaging to produce an address which may be linear, char, or sprite.

Linear Mode:

* DST acts as a base address for the current byte.
* The true address == DST + currentCount..
* 256 bytes max, means 512 pixels wide.
* When the instruction completes, DST += currentCount.

Char Mode

* DST acts as a base-address for the current row.
* The true address == DST + (currentCount>>2) <<5 + (currentCount&3)
* 256 bytes max, means 512 pixels wide, or 64 chars wide.
* When the instruction completes, DST += 4.

Sprite Mode

* DST acts as a base-address for the current row.
* The true address == DST + (currentCount>>3) <<6 + (currentCount&7)
* 256 bytes max, means 512 pixels wide, or 32 sprites wide.
* When the instruction completes, DST += 8.

There is a ‘direction’ which is extracted from the parameter.

There is a ‘maxCount’, which is extracted from the parameter.

There is a ‘currentCount’ which begins at 0, and counts to maxCount.

There is a ‘currentAddress’ which is generated from the ‘currentCount’ and the ‘direction’.

##### PIXELSHIFT Right of a 4-byte Block:

BEFORE: AB CD EF GH

AFTER: HA BC DE FG

; RD WR STATE

R[3] EDGE <= DATA\_IN.H ; \_H \_R1, \_S1 (PREFETCH)

R[0] H <= EDGE L <= DATA\_IN.H EDGE <= DATA\_IN.L ; AB \_R2, \_S2 (FETCH)

W[0] {H,L} ; HA \_W (STORE)

R[1] H <= EDGE L <= DATA\_IN.H EDGE <= DATA\_IN.L ; CD \_R2, \_S2 (FETCH)

W[1] {H,L} ; BC \_W (STORE)

R[2] H <= EDGE L <= DATA\_IN.H EDGE <= DATA\_IN.L ; EF \_R2, \_S2 (FETCH)

W[2] {H,L} ; DE \_W (STORE)

R[3] H <= EDGE L <= DATA\_IN.H EDGE <= DATA\_IN.L ; GH \_R2, \_S2 (FETCH)

W[3] {HL} ; FG \_W (STORE)

##### PIXELSHIFT Left of a 4-byte Block:

BEFORE: AB CD EF GH

AFTER: BC DE FG HA

; RD WR STATE

R[0] EDGE <= DATA\_IN.L ; A\_ \_R1, \_S1 (PREFETCH)

R[3] L <= EDGE H <= DATA\_IN.L EDGE <= DATA\_IN.H ; GH \_R2, \_S2 (FETCH)

W[3] {H,L} ; HA \_W (STORE)

R[2] L <= EDGE H <= DATA\_IN.L EDGE <= DATA\_IN.H ; EF \_R2, \_S2 (FETCH)

W[2] {H,L} ; FG \_W (STORE)

R[1] L <= EDGE H <= DATA\_IN.L EDGE <= DATA\_IN.H ; CD \_R2, \_S2 (FETCH)

W[1] {H,L} ; DE \_W (STORE)

R[0] L <= EDGE H <= DATA\_IN.L EDGE <= DATA\_IN.H ; AB \_R2, \_S2 (FETCH)

W[0] {HL} ; BC \_W (STORE)

#### ADDSTOD: Add SRC To DST (Extended Instruction)

[NOT IMPLEMENTED]

* 00111110 00000100

#### ADDDTOS: Add DST To SRC (Extended Instruction)

[NOT IMPLEMENTED]

* 00111110 00000101

#### ADDNTOS: Add n To SRC (Extended Instruction)

[NOT IMPLEMENTED]

* 00111110 0000011S NNNNNNNN  
  S: 0=Unsigned, 1=Signed  
  N: 0..255, or -128..127

#### ADDNTOD: Add N To DST (Extended Instruction)

[NOT IMPLEMENTED]

* 00111110 0000100S NNNNNNNN  
  S: 0=Unsigned, 1=Signed  
  N: 0..255, or -128..127

#### REPEAT n (Extended Instruction) [TEST]

Begin a repeat block. All instructions within the repeat block will be repeated n+1 times.

* 00111110 00001010 NNNNNNNN

The copper sets its' internal repeatCounter register to the value of N, and remembers the IP address that immediately follows this instruction in the repeatIP register. An EndRepeat instruction may cause the copper to return to that address.

NOTE: You cannot nest REPEAT blocks, because the copper has no concept of a stack.

#### EndRepeat (Extended Instruction) [TEST]

Ends a repeat block. If the repeatCounter register is non-zero, it will be decremented, and the copper's IP will be sent back to the address stored in repeatIP. If the repeatCount is zero, then the instruction acts like a NOP, and no branch is taken.

* 00111110 00001011

*NOTE: There is a 4 cycle penalty each time the copper executes an ENDREPEAT instruction, so loop unrolling is something to consider, if you are counting cycles.*

#### Unused OpCodes (Extended Instructions)

* 00111110 00001101, up to 00111110 11111111

#### INT: INTERRUPT (Extended Instruction)

Cause a copper-based interrupt to occur.

* 00111110 00001100

##### STOP: Stop Copper

The Copper’s ENABLE flag returns to 0, and the Copper’s IP remains pointing to the next instruction.

The Z80 could re-enable the copper, and it would continue from where it left off.

The Z80 could re-enable the copper, and include a RESTART flag, to cause it to re-run from the COPPER\_START\_ADDR.

* 00111111 (OPCODE)

NOTE: Since STOP clears the enable flag, the copper will NOT auto-restart on the next VBLANK unless the Z80 sets the enable flag back to TRUE first, which in this case (after a STOP), would immediately allow the copper to begin executing instructions again. If you want the copper list execution to stop, but not stop the copper itself, use a WAITS 255, which causes the copper to wait for a scanline that will never occur. Thus, it will never get a chance to execute instructions again, for the remainder of this frame. And, assuming AUTO\_RESTART is enabled, the copper would reset to the top of the instruction list as soon as a VBLANK does occur.

#### 010 Instructions

##### DMAS: DMA 1..32 bytes from SRC to DST in VRAM

* 010NNNNN (OPCODE)
* N: 0..31 Copy (1..32) bytes from SRC to DST.

NOTE: If you, at runtime, want to adjust the value to 0, then replace the instruction with a SKIP 0 instruction.

#### 011 Instructions

##### DMAL: DMA 1..8192 bytes from SRC to DST in VRAM

Copies N bytes from SRC to DST, with auto-inc/dec.

Good for large block moves and copies, such as BG map scrolling.

011NNNNN NNNNNNNN (OPCODE, PARAM)

* N: 0..8191 Copy (1…8192) bytes from SRC to DST.

NOTE: If you, at runtime, want to adjust the value to 0, then replace the instruction with a SKIP 1 instruction.

### 1 Instructions

#### SKIP: Skip #bytes forward in copper list

(0 is also a valid # of bytes to skip! Essentially it becomes a NOP)

* 1SSSSSSS (OPCODE)
* S = 0..127, #bytes to skip forward in Copper list

# Internal Functional Overview

The Copper has the following major interconnect blocks:

* CPU Bus
* VRAM Bus
* Video Signal Inputs

The Copper offers a few read/write registers through the CPU Bus.

Internally, it runs a state machine that monitors the video signals, and produces VRAM Bus requests in order to move memory around and fetch its’ own instructions and parameters.

The VRAM Bus requests may be denied if the CPU is using the VRAM Bus on the same cycle. In this case, the Copper state machine cannot advance, and the same state executes next clock. Therefore, it waits until the request is accepted. The copper is designed to allow this to happen as frequently as wanted, including many times sequentially. In practice, the Z80 CPU makes very infrequent accesses, and the copper only uses the bus roughly every other cycle at most, so it should lose very few cycles to CPU activity.

# Copper Compiler

A copper list is stored in a file with the extention .COP, and is compiled into binary (.BIN) form and a listing (.LST) form, using a script called CopperCompiler.py.

## Script Language

The lines that are written in the script are intended to be more readable than the actual instructions are. When the script is compiled, it produces the raw binary data, but it also produces a listing which shows the actual decoded instructions as well.

Values can be expressed in binary, hex, or decimal. They can end with an H or L, to indicate high or low byte value.

[0<b|d|x>]<n>

RGB colors can be generated by using RGB(r,g,b) where r, g, and b are 4bit values.

RGB(<R>,<G>,<B>)[H|L]

Comments begin with a #, anywhere on a line.

Following, are a list of all the instructions, and how they are written in the .COP file.

### SETADDR

<SRC|DST>=<BANK>:<ADDR>[,PostInc|PostDec][,PostReset]

Eg. SRC = 0x6 : 0x0100, PostINC, PostReset

### FILLV

FILL <COUNT> @ DST = <VALUE>

COUNT = 1...32

VALUE = -128…255

Eg. FILL\_BYTES 25 @ DST = 10

### WAITNSS, WAITNSL

WAIT\_NUM\_SCANLINES[.<S|L>] <COUNT>

COUNT = -512…511

### WAITS

WAIT\_UNTIL\_SCANLINE <SCANLINE>

SCANLINE = -256…255

### STOP

STOP

### DMAS, DMAL

DMA[.<S|L>] <COUNT>

COUNT = 1…8192

### SKIP

SKIP <COUNT>

COUNT = 0..127

## Example.cop